(NASA-CR-163425) FLIGHT SOFTWARE REQUIREMENTS AND DESIGN SUPPORT SYSTEM Final Report (Colorado Univ. at Boulder.) 18 p HC A02/MF A01 CSCL 09B

N80-30061

Unclas
G3/61 28333

Final Report

To National Aeronautics and Space Administration
Langley Research Center

For Grant NSG 1638
Flight Software Requirements and Design Support System

William Riddle
Bryan Edwards

August 1980

Introduction

The overall intent of this project was to investigate the desirability and feasibility of computer-augmented support for the pre-implementation activities occurring during the development of flight control software. The specific topics to be investigated were:

- the capabilities to be included in a pre-implementation support system for flight control software system development, and
- the specification of a preliminary design for such a system.

Further, the pre-implementation support system was to be characterized and specified under the constraints that it:

- support both description and assessment of flight control software requirements definitions and design specification,
- account for known software description and assessment techniques,
- be compatible with existing and planned NASA flight control software development support systems, a

- does not impose, but may encourage, specific development methodologies.

In this final report, we give an overview of the results obtained during the project -- specific details are given in the reports included as appendices. In the next section, we address the issues concerning the languages provided for the description of requirements and designs. Then, in the succeeding sections, we address issues concerning the tools included in the pre-implementation support system and the basic nature of the support system itself.

Languages

Our investigation of the capabilities desirable for the description of flight control software during pre-implementation development phases began with an attempt to describe the Annular Suspension Pointing System in a fairly rigorous, formal manner. In reading the original description of ASPS, which had been prepared by NASA Langley personnel, we felt that it suffered the usual problems encountered when using informal description media such as English prose, block diagrams and timing diagrams. The description itself had very little structure and was relatively hard to follow. Also, various aspects of the system were presented in an inconsistent manner, both in terms of the content of the description and in terms of the level of the description, and it was difficult to detect inconsistencies and omissions.

Using the understanding of ASPS obtained from the original description, another description was produced (see Appendix A)

using the DREAM Design Notation (DDN). The DDN description is hierarchically structured and at each level of the hierarchy, all components of the system are presented at the same level of detail. Further, in the DDN description, all relationships among system components are explicitly stated and thus inconsistencies and omissions are more easily observed.

The DDN description is decidedly better in terms of rigor. However, it fails to capture several aspects of the ASPS system: absolute time characteristics of processes within the system, time slicing of processes, system initialization, and the use of the main processor interval timer to verify the master timing pulse. Failure to capture some of these aspects is acceptable because they are not really concerns during requirements definition or (high-level) design. However, the inability to describe them highlights a (previously known) failing of the DDN language, namely that it does not contain facilities for describing the absolute time characteristics of systems.

In order to more completely consider the issue of the rigorous specification of requirements and designs, we wanted to prepare other descriptions of ASPS in other well-defined design description languages. We attempted a description in the Gypsy language, but did not find it suitable for describing ASPS at the design or requirements definition level -- the level at which the system was described in DDN.

The difficulty lay in some major differences between the DDN and Gypsy languages. These two languages differ significantly in

terms of two characteristics which may be called property scope and property statement. The property scope characteristic concerns whether or not the global, overall properties of a system may be described -- a language having local property scope may be used to describe the characteristics of a system's components and one having global property scope may be used to state the properties of collections of components. The property statement characteristic concerns how system properties are described. A language having implicit property statement, with respect to some property, allows that property to be only implicitly stated, i.e., the property must be deduced from the information which is explicitly stated. A language having explicit property statement, on the other hand, allows that property to be stated using the primitive constructs of the language.

In attempting to develop a Gypsy description of ASPS that was comparable to the DDN description, we found that whereas DDN has both local and global property scope, Gypsy has only local property scope, and whereas DDN has explicit property statement with respect to behavioral properties, Gypsy has only implicit property statement with respect to behavioral properties. Because of the impossibility of describing global properties and explicitly describing behavioral properties we did not feel it was fruitful to further investigate the Gypsy language as a medium for the description of flight control software requirements and designs. Specifically, as a result of this exercise, we feel that any language for the pre-implementation description of flight control software should contain capabilities similar in nature to the following which are present in the DDN

language:

- SUBCOMPONENTS -- to allow the explicit description of the hierarchical organization of the system
- CONNECTIONS -- to allow the explicit specification of the global connectivity relationships among a system's components
- EVENTS and DESIRED BEHAVIOR -- to allow the explicit description of the global behavior characteristics to be exhibited by the system

We feel that it is more than merely desirable to be able to state the global properties of a system during its design since these global property statements may be effectively used to check the suitability of lower-level design decisions made during the process of iterative enhancement. This ability in turn helps in verifying that the design is internally consistent and permits the early discovery of omissions.

We came to one other conclusion as a result of our attempts to use the Gypsy language to describe ASPS. We feel that Gypsy is more appropriately considered to be an implementation description medium rather than a pre-implementation description medium. Further we feel that the fact that the Gypsy language utilizes basic concepts (such as message transfer) which makes it a natural companion to the DDN language. There are other natural companion languages of course, such as Ada, but the point is that in developing pre-implementation languages, they should be based upon the same set of primitive concepts as the implementation languages, and vice versa.

To broaden our investigation of languages despite our inability to use the Gypsy language in this endeavor, we turned to a survey

of actual descriptions of flight control software systems. We used the Proceedings of the Programming Languages for Real-time Systems Workshop held at NASA Langley Research Center in October 1979 and scanned these proceedings to determine answers to the following questions:

- What aspects of flight control systems are typically described in requirements and design descriptions and what aspects are frequently omitted from these descriptions?
- What description media are typically used?
- What description media are not used and could they be effectively used?
- What is a typical description medium used to describe?
- What could a particular description medium be fruitfully used to describe?

We summarize our findings in the following paragraphs.

The most commonly used descriptive medium was English prose structured in a hierarchical or outline form. It was usually used to describe the behavior of the system is a very loose manner. One descriptive technique that could have been, but was not, used to make these behavior descriptions a bit more rigorous is regular expressions.

Block diagrams and flow charts were also used. Block diagrams were used to describe the functional units of a system. Very frequently, the units were physical, hardware ones, and the lines between the blocks represented communication pathways. Usually the paths in the block diagrams were bidirectional. Flow charts were used to describe some type of cycle, for example, an execution cycle or a project life cycle. Usually, but not always, the arcs in the

flow charts represented directed paths.

A variety of information was represented in tabular form. Included were tables of items and their associated costs, cross-correlation tables and transition tables. A regular expression notation could have been used to effectively represent both the information presented in the tables and correlations of items indicated in the tables, e.g., correlations among transitions indicating common transition sequences.

Overall system timing was usually presented in the form of timing diagrams. This information could have been more extensively and formally represented by using clock variables (such as found in Gypsy) and giving regular expressions over the values of these variables to express timing constraints.

Graphs indicating the relationships between two variables were frequently used to indicate project dynamics. These graphs usually communicated a history of the occurrences of an event over time.

Lastly, schematic diagrams were frequently used to give a drawing of actual equipment or an "artist's conception" of the final system. Sometimes these schematic diagrams were presented with associated block diagrams to show the physical layout of a system's functional units.

The descriptions frequently did not distinguish between whether a particular aspect of the system was a system requirement or part of the system design. This is a very common happening when the description is prepared after the fact and the developers

have lost sight of whether a particular aspect is there because it was part of the initial requirements or because it was the result of a design decision. It points out the critical need to carefully demarcate the description of a system's requirements, which are immutable and fixed, and the description of the system's design, which may be changed as long as the new system design still leads to the delivery of the characteristics and properties specified, directly or indirectly, in the requirements.

Another characteristic of the descriptions inspected was that it was difficult to trace the effect of decisions through the design and relate these decisions to aspects of the final system. It would seem that, in addition to facilities for directly capturing the effect of a decision, it would be important to have the capability to have more structured descriptions that would capture, in the organization of the description itself, some of the interrelationships among the various aspects of the description. Also, it is important for the purpose of tracing requirements and design decisions to be able to rigorously specify the "world" in which the eventual system will operate since the hardware, software and human aspects of this "world" impact and constrain the decisions that can be made.

Tools

Some attempt was made to identify the tools that could and should be included in a pre-implementation support system oriented towards flight control software system development. One attack on this aspect was to hypothesize the sequence of activities that must be carried

out by a number of agents during flight control software system development -- in essence, this amounted to giving a very detailed life-cycle for the development of these systems. Once this was developed, note was made of the tools that would be useful in aiding each of the activities. The results of this part of the project are given in Appendix B.

To augment this activity and give some indication of the spectrum of tools which are available (as opposed to those which should be available), a bibliography was prepared giving references to literature concerned with tools which could support pre-implementation development activities. This bibliography appears as Appendix C. The bibliography was not intended to be complete but a number of sources were utilized and it does give a fairly accurate indication of the range of capabilities available. (It should be emphasized that the bibliography, besides fixing on tools for aiding pre-implementation activities, does not reference any literature on development methods or techniques, i.e., cognitive tools, or any literature on descriptive media, i.e., notational tools.)

The conclusion that we wish to suggest may be drawn from a perusal of the tools bibliography is that a large number of quite adequate tools exist already and that an effective and useful collection of techniques and capabilities can easily be identified. The problem that arises is obvious -- although the collection of capabilities and techniques may easily be identified, implementing them in some coherent, integrated manner is entirely a different matter, let alone very difficult. We will address this issue again in the next section.

We were unable to give concentrated attention to the very important question of the compatibility of capabilities and techniques already present in the MUST environment and those which should be included in the pre-implementation support system. It is clear that the intent of some of the already existing capabilities and techniques is consonant with the requirements levied upon the capabilities and techniques necessary and desirable during pre-implementation activities. This does not mean, however, that the existing techniques and capabilities are the ones of choice when efficiency, effectiveness, and performance criteria are considered.

For example, the DAVE system employs annotated graph representations of a program and then does graph searching activities to identify anomalous occurrences. This processing can also be thought of as doing language theoretic operations upon sets of sequences defined by regular expressions; this latter, functionally equivalent processing approach was not chosen for the DAVE system for efficiency and performance reasons. It is, for a number of reasons, quite viable at the pre-implementation level because of the generally smaller and less complex descriptions (because of the use of abstraction) which exist at this level.

The conclusion which we reach from this train of thought is that the questions of what capabilities and techniques should be used during pre-implementation activities and the extent of overlap between the capabilities and techniques used before and during implementation are much more difficult than originally perceived. The more we addressed these questions, the more we felt that we could not, in

the lifetime of this project, adequately address them.

We did, however, develop the strong feeling that the way to approach these questions was experimentally. By this we mean that a trial-and-error, iterative approach to deciding the applicability of already existing capabilities and techniques and deciding the relative effectiveness of new techniques and capabilities is the appropriate way to proceed. In the next section, we will give some quidelines for carrying out this approach -- those quidelines were in part affected by our consideration of the questions discussed here.

In concluding this section, we should emphasize that the tools of use during pre-implementation are, in a sense, fundamentally different from those of which aid implementation activities. By nature, the activities occurring during pre-implementation are exploratory and speculative; whereas those occurring during implementation are straightforward and well-defined. It is, therefore, hard to duplicate the level of automation that has been achieved for tools of use during implementation. Instead, it is necessary, at this point in time, to make the tools provided to support pre-implementation activities very interactive -- they should in essence be considered as extensions of the human developers and should be viewed as augmenting rather than replacing the developers. Tools which serve to "animate" the system description and provide information concerning its behavior as feedback to the developers are particularly important in this regard.

Environments

Tools become all the more useful and effective when they are provided as a coherent, well-integrated set -- one has only to compare a toolbox with a machine shop to come to this conclusion. The result of this integration is an environment in which the development practitioners may perform their day-to-day work. To conclude the investigations of this project, we were interested in the questions: what is a good basis for providing an integrated set of tools for aiding flight control software development and how can such an environment be delivered to development practitioners in a reasoned and relatively inexpensive manner?

With regard to the first question we feel that the organization of choice for development environments is one in which there is a central data base which serves as a repository for all information ever generated about the system under development. This is not a particularly startling observation at this point in time as this organization is the one most often used or suggested. We would like to point out, however, that it is about the only organization that is consistent with our suggestion that the investigation of tools be experimental in nature. This is because it provides the flexibility necessary to add and subtract tools since it imposes no restrictions on what tools are available and how they are provided in the environment. (Additionally, it should be noted that this organization permits avoiding the imposition of methodologies through the environment itself since methodological constraints come, when using this organization, in the form of rules and guidelines for using the

tools provided by the environment.)

We have reached this conclusion as a result of a number of introspective reviews of our previous work on the DREAM development support system. One of these reviews was very general in nature and tried to assess the current state of affairs accounting for the history of development environments in general. This review appears as Appendix D.

We also have prepared a review -- appearing as Appendix E -- which considers the DREAM system alone, but pays attention to non-technical as well as technical aspects. As a result of this review, we feel strongly that the DREAM system organization and the set of concepts embodied in the DDN language are the right way to proceed but that the set of concepts is incomplete (as noted before in our discussion of language issues) and that more careful thought is needed concerning the syntax of the language delivering the concepts and the organization of the tools in the environment.

The final review appears as Appendix F and contains little hindsight but rather attempts to establish a vocabulary for talking about environments and a framework for thinking about their design and implementation.

Another aspect of this part of the project was to prepare a bibliography on software development environments -- this appears as Appendix G. Again, an attempt was made to be fairly complete but the bibliography was not intended to be exhaustive of the literature. It was, however, extremely disconcerting to find two

other bibliographies that had very little overlap with ours -- one which covered the area of programming environments such as the Lisp machine and the other covered development environments. (Our bibliography has since been merged with the second one cited above and will appear in the Proceedings of the Symposium on Software Engineering Environments held in Cologne, Germany, which is to be published by North Holland in October.) The lesson to be learned from finding these other bibliographies is that there is a tremendous amount of activity in the area of environments and that the literature is not appearing in a small, concentrated segment of the computer science publications.

To conclude, we would like to suggest an incremental approach to delivering flight software development support environments in which a series of progressively more sophisticated environments are produced, the last of which is what we feel would be the complete environment. (It should be noted that we were helped in developing this incremental approach by participating in a NBS-sponsored workshop at which this topic was given concentrated attention by a working group of seven people over three days.)

The initial environment would provide the minimum necessary support, much of it through manual rather than automated procedures. This system (and all of the others) assumes the presence of several pieces of standard system software (such as compilers, linking loaders, runtime libraries, file systems, etc.) such as are found in the MUST environment. To arrive at a minimal pre-implementation environment, little need be added to this assumed core since a number

of manual procedures are available for the control of pieces of textual information and the assessment of its validity. The point is that a minimal environment can be made available by augmenting MUST with manual procedures for requirements definition, design, testing and project management.

Requirements would be handled by instituting manual procedures (for example, the use of SADT diagrams) for the definition of requirements. Design could be handled by informal design procedures chosen and implemented by the project leader. To handle the organizing of the various descriptions, a partially implemented text control system (using the ideas embodied in the UNIX source code control system) would be included. This system would include facilities for text entry and editing and facilities for "version" control. It would also include facilities for maintaining a simple directory allowing for the easy retrieval of pieces of text.

Analysis would be handled manually as would be project management. This latter aid could be augmented by facilities for the preparation of PERT-type charts.

This first in the series of environments is admittedly primitive and simple -- but it indicates that a good deal of aid could be provided by a relatively simple extension to the MUST environment. It relies upon existing software and manual procedures and, as such, does not represent a large expense in terms of time or resources in order to provide a basic, simple environment. We feel that this environment would be sufficient to support small, 2-5 person flight control software development projects.

Extensions can be made to arrive at a second system in the series. The most extensive addition would be a data base system (mostly already provided in MUST) which would provide support for keeping track of objects, object attributes, and relationships among objects. This "simple" change has very broad implications since it moves the environment towards one in which there is a central repository of information and thus the basis for tool integration.

Requirements could now be kept in machine-processable form in terms of objects, attributes and relations (much as in the ISDOS system). Simple analysis procedures could be provided to analyze requirement descriptions for completeness and form. Manual procedures could be defined for more extensive analysis.

The manual design procedures could also be replaced by simple, formal techniques which also relied on the definition of designs in terms of objects, attributes and relations. As with requirements, the designs could be analyzed for completeness and form. However, automated procedures for checking the consistency of requirements definitions and designs would not be included.

Project management could also be aided with the addition of a simple project control system, again relying on the use of the object-attribute-relation data base. Automated aids for the generation of project status reports, milestone progress reports and dependency charts could also be included.

This second environment in the series would not be a terribly

large step away from the first and would not be difficult to implement. But it lays the groundwork for all subsequent environments through the introduction of the central data base. We feel that it could be effective in aiding medium sized projects.

The rest of the environments in the series would be obtained by adding more and more sophisticated tools. Languages could easily be added to the system and the data base could be used to help maintain descriptions written within the languages since a fragment of text can be viewed as a object and the means exists for keeping track of these text fragments, their attributes and their relationships.

Tools could rather easily be added or deleted from the environment as long as each tool is viewed as using the information in the data base to produce new information to be added to the data base. Tools for possible inclusion would be: data flow analyzers, pretty printers, flow charters, control flow analyzers, performance monitors, simulators, cross referencers, etc. In fact, most of the tools indicated in the tools bibliography appearing in the appendices are candidates with the primary decision being whether the effort of implementing versions which operate on the descriptions in the languages provided by the environment is cost effective with respect to the benefit derived.

This scenario of successively more sophisticated environments is simple but effective. It allows a gradual committment to the production of an environment and a gradual expenditure of effort and money. It also provides useful environments along the way --

environments which are not only able to be used for flight control software development but also may be used to evaluate the effectiveness of environments in the flight control software development situation and thus the efficacy of proceeding farther through the series.

Conclusion

We have indicated some of the results obtained during through this project. The details of the results are reported in the appendices and we have here provided just and overview and given the conclusions which we feel may be drawn. In summary, we feel that the means and techniques already exist for the preparation of flight control software development support systems but that the only way to effectively determine what should be in the support system and just how effective the support system will be is by an incremental, experimental approach. We have provided a game-plan for such an approach and given advice on what should be considered as candidate tools for inclusion in the successive versions of the support system produced by following this approach.